Neural Network Vectorization

In this section, we derive a vectorized version of our neural network. In our earlier description of Neural Networks, we had already given a partially vectorized implementation, that is quite efficient if we are working with only a single example at a time. We now describe how to implement the algorithm so that it simultaneously processes multiple training examples. Specifically, we will do this for the forward propagation and backpropagation steps, as well as for learning a sparse set of features.

Forward propagation

Consider a 3 layer neural network (with one input, one hidden, and one output layer), and suppose `x` is a column vector containing a single training example $x^{(i)} \in \Re^n$. Then the forward propagation step is given by:

$$\begin{aligned} z^{(2)} &= W^{(1)} x + b^{(1)} \\ a^{(2)} &= f(z^{(2)}) \\ z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)} \\ h_{W,b}(x) &= a^{(3)} = f(z^{(3)}) \end{aligned}$$

This is a fairly efficient implementation for a single example. If we have m examples, then we would wrap a `for` loop around this.

Concretely, following the Logistic Regression Vectorization Example, let the Matlab/Octave variable `x` be a matrix containing the training inputs, so that `x(:,i)` is the i-th training example. We can then implement forward propagation as:

```
% Unvectorized implementation
for i=1:m,
  z2 = W1 * x(:,i) + b1;
  a2 = f(z2);
  z3 = W2 * a2 + b2;
  h(:,i) = f(z3);
end;
```

Can we get rid of the `for` loop? For many algorithms, we will represent intermediate stages of computation via vectors. For example, `z2`, `a2`, and `z3` here are all column vectors that're used to compute the activations of the hidden and output layers. In order to take better advantage of parallelism and efficient matrix operations, we would like to *have our algorithm operate* *simultaneously on many training examples*. Let us temporarily ignore `b1` and `b2` (say, set them to zero for now). We can then implement the following:

```
% Vectorized implementation (ignoring b1, b2)
z2 = W1 * x;
a2 = f(z2);
z3 = W2 * a2;
h = f(z3)
```

In this implementation, `z2`, `a2`, and `z3` are all matrices, with one column per training example. A common design pattern in vectorizing across training examples is that whereas previously we had a column vector (such as `z2`) per training example, we can often instead try to compute a matrix so that all of these column vectors are stacked together to form a matrix. Concretely, in this example, `a2` becomes a s_2 by $m$matrix (where s_2 is the number of units in layer 2 of the network, and m is the number of training examples). And, the i-th column of `a2`contains the activations of the hidden units (layer 2 of the network) when the i-th training example `x(:,i)` is input to the network.

In the implementation above, we have assumed that the activation function `f(z)` takes as input a matrix `z`, and applies the activation function component-wise to the input. Note that your implementation of `f(z)` should use Matlab/Octave's matrix operations as much as possible, and avoid `for` loops as well. We illustrate this below, assuming that `f(z)` is the sigmoid activation function:

```
% Inefficient, unvectorized implementation of the
activation function
function output = unvectorized_f(z)
output = zeros(size(z))
for i=1:size(z,1),
  for j=1:size(z,2),
    output(i,j) = 1/(1+exp(-z(i,j)));
  end;
end;
end

% Efficient, vectorized implementation of the activation
function
function output = vectorized_f(z)
output = 1./(1+exp(-z));     % "./" is Matlab/Octave's
element-wise division operator.
end
```

Finally, our vectorized implementation of forward propagation above had ignored b1 and b2. To incorporate those back in, we will use Matlab/Octave's built-in repmat function. We have:

```
% Vectorized implementation of forward propagation
z2 = W1 * x + repmat(b1,1,m);
a2 = f(z2);
z3 = W2 * a2 + repmat(b2,1,m);
h = f(z3)
```

The result of repmat(b1,1,m) is a matrix formed by taking the column vector b1 and stacking m copies of them in columns as follows

$$\begin{bmatrix} | & | & & | \\ \mathrm{b1} & \mathrm{b1} & \cdots & \mathrm{b1} \\ | & | & & | \end{bmatrix}.$$

This forms a s_2 by m matrix. Thus, the result of adding this to W1 * x is that each column of the matrix gets b1 added to it, as desired. See Matlab/Octave's documentation (type "help repmat") for more information. As a Matlab/Octave built-in function, repmat is very efficient as well, and runs much faster than if you were to implement the same thing yourself using a for loop.

Backpropagation

We now describe the main ideas behind vectorizing backpropagation. Before reading this section, we strongly encourage you to carefully step through all the forward propagation code examples above to make sure you fully understand them. In this text, we'll only sketch the details of how to vectorize backpropagation, and leave you to derive the details in the Vectorization exercise.

We are in a supervised learning setting, so that we have a training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples. (For the autoencoder, we simply set $y^{(i)} = x^{(i)}$, but our derivation here will consider this more general setting.)

Suppose we have s_3 dimensional outputs, so that our target labels are $y^{(i)} \in \Re^{s_3}$. In our Matlab/Octave datastructure, we will stack these in columns to form a Matlab/Octave variable y, so that the i-th column y(:,i) is $y^{(i)}$.

We now want to compute the gradient terms $\nabla_{W^{(l)}} J(W,b)$ and $\nabla_{b^{(l)}} J(W,b)$. Consider the first of these terms. Following our earlier description of the Backpropagation Algorithm, we had that for a single training example (x,y), we can compute the derivatives as

$$\begin{aligned}
\delta^{(3)} &= -(y - a^{(3)}) \bullet f'(z^{(3)}), \\
\delta^{(2)} &= ((W^{(2)})^T\delta^{(3)}) \bullet f'(z^{(2)}), \\
\nabla_{W^{(2)}} J(W,b;x,y) &= \delta^{(3)} (a^{(2)})^T, \\
\nabla_{W^{(1)}} J(W,b;x,y) &= \delta^{(2)} (a^{(1)})^T.
\end{aligned}$$

Here, $\bullet$ denotes element-wise product. For simplicity, our description here will ignore the derivatives with respect to $b^{(l)}$, though your implementation of backpropagation will have to compute those derivatives too.

Suppose we have already implemented the vectorized forward propagation method, so that the matrix-valued `z2`, `a2`, `z3` and `h` are computed as described above. We can then implement an *unvectorized* version of backpropagation as follows:

```
gradW1 = zeros(size(W1));
gradW2 = zeros(size(W2));
for i=1:m,
  delta3 = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
  delta2 = W2'*delta3(:,i) .* fprime(z2(:,i));

  gradW2 = gradW2 + delta3*a2(:,i)';
  gradW1 = gradW1 + delta2*a1(:,i)';
end;
```

This implementation has a `for` loop. We would like to come up with an implementation that simultaneously performs backpropagation on all the examples, and eliminates this `for` loop.

To do so, we will replace the vectors `delta3` and `delta2` with matrices, where one column of each matrix corresponds to each training example. We will also implement a function `fprime(z)` that takes as input a matrix `z`, and applies $f'(\cdot)$ element-wise. Each of the four lines of Matlab in the `for` loop above can then be vectorized and replaced with a single line of Matlab code (without a surrounding `for` loop).

In the Vectorization exercise, we ask you to derive the vectorized version of this algorithm by yourself. If you are able to do it from this description, we strongly encourage you to do so. Here also are some Backpropagation vectorization hints; however, we encourage you to try to carry out the vectorization yourself without looking at the hints.

Sparse autoencoder

The sparse autoencoder neural network has an additional sparsity penalty that constrains neurons' average firing rate to be close to some target activation ρ. When performing backpropagation on a single training example, we had taken into the account the sparsity penalty by computing the following:

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(- \frac{\rho}{\hat\rho_i} + \frac{1-\rho}{1-\hat\rho_i} \right) \right) f'(z_i^{(2)}) .$$

In the *unvectorized* case, this was computed as:

```
% Sparsity Penalty Delta
sparsity_delta = - rho ./ rho_hat + (1 -
rho) ./ (1 - rho_hat);
for i=1:m,
  ...
  delta2 = (W2'*delta3(:,i) +
beta*sparsity_delta).* fprime(z2(:,i));
  ...
end;
```

The code above still had a `for` loop over the training set, and `delta2` was a column vector.

In contrast, recall that in the vectorized case, `delta2` is now a matrix with m columns corresponding to the m training examples. Now, notice that the `sparsity_delta` term is the same regardless of what training example we are processing. This suggests that vectorizing the computation above can be done by simply adding the same value to each column when constructing the `delta2` matrix. Thus, to vectorize the above computation, we can simply add `sparsity_delta` (e.g., using `repmat`) to each column of `delta2`.

Exercise:Vectorization

Contents

[hide]

Vectorization

In the previous problem set, we implemented a sparse autoencoder for patches taken from natural images. In this problem set, you will vectorize your code to make it run much faster, and further adapt your sparse autoencoder to work on images of handwritten digits. Your network for learning from handwritten digits will be much larger than the one you'd trained on the natural images, and so using the original implementation would have been painfully slow. But with a vectorized implementation of the autoencoder, you will be able to get this to run in a reasonable amount of computation time.

Support Code/Data

The following additional files are required for this exercise:

- MNIST Dataset (Training Images)
- MNIST Dataset (Training Labels)
- Support functions for loading MNIST in Matlab

Step 1: Vectorize your Sparse Autoencoder Implementation

Using the ideas from Vectorization and Neural Network Vectorization, vectorize your implementation of `sparseAutoencoderCost.m`. In our implementation, we were able to remove all for-loops with the use of matrix operations and `repmat`. (If you want to play with more advanced vectorization ideas, also type `help bsxfun`. The `bsxfun` function provides an alternative to `repmat` for some of the vectorization steps, but is not necessary for this exercise). A vectorized version of our sparse autoencoder code ran in under one minute on a fast computer (for learning 25 features from 10000 8x8 image patches).

(Note that you do not need to vectorize the code in the other files.)

Step 2: Learn features for handwritten digits

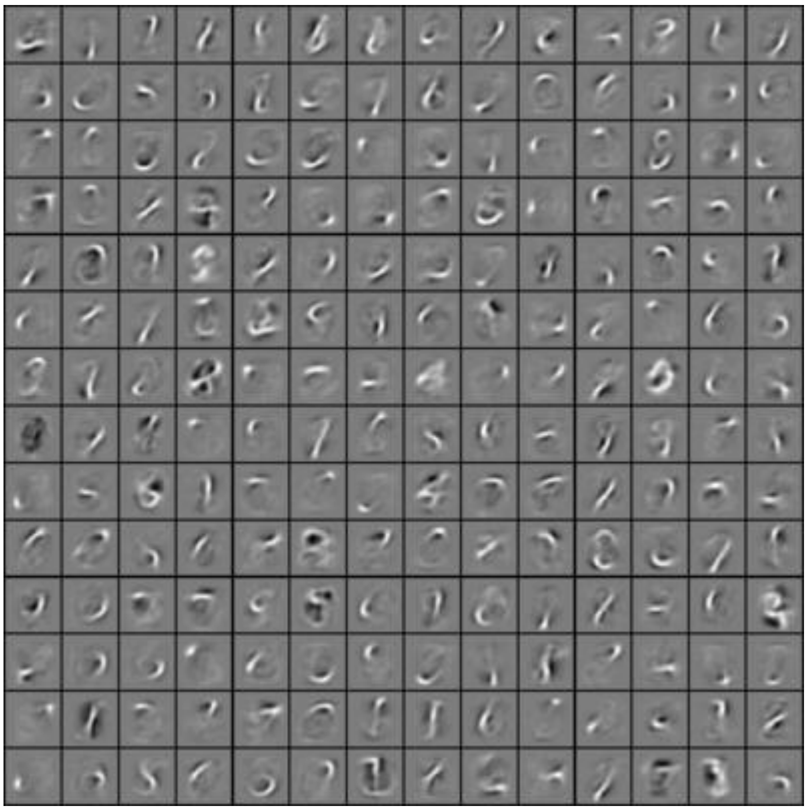
Now that you have vectorized the code, it is easy to learn larger sets of features on medium sized images. In this part of the exercise, you will use your sparse autoencoder to learn features for handwritten digits from the MNIST dataset.

The MNIST data is available at [1]. Download the file `train-images-idx3-ubyte.gz` and decompress it. After obtaining the source images, you should use helper functions that we provide to load the data into Matlab as matrices. While the helper functions that we provide will load both the input examples x and the class labels y, for this assignment, you will only need the input examples x since the sparse autoencoder is an *unsupervised* learning algorithm. (In a later assignment, we will use the labels y as well.)

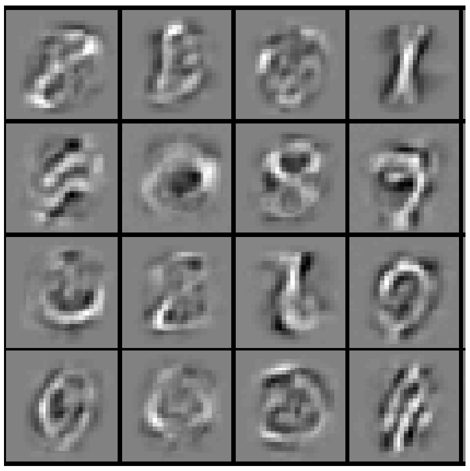
The following set of parameters worked well for us to learn good features on the MNIST dataset:

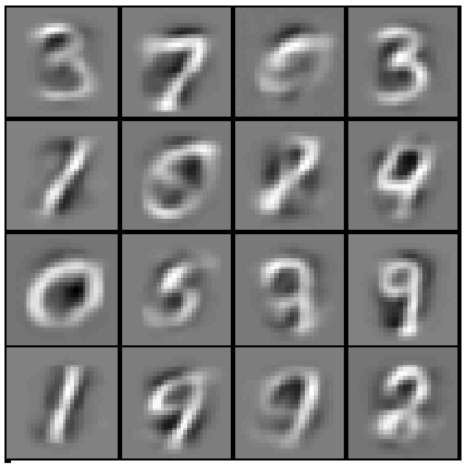
```
visibleSize = 28*28
hiddenSize = 196
sparsityParam = 0.1
lambda = 3e-3
beta = 3
patches = first 10000 images from the MNIST dataset
```

After 400 iterations of updates using minFunc, your autoencoder should have learned features that resemble pen strokes. In other words, this has learned to represent handwritten characters in terms of what pen strokes appear in an image. Our implementation takes around 15-20 minutes on a fast machine. Visualized, the features should look like the following image:



If your parameters are improperly tuned, or if your implementation of the autoencoder is buggy, you may get one of the following images instead:





If your image looks like one of the above images, check your code and parameters again. Learning these features are a prelude to the later exercises, where we shall see how they will be useful for classification.